Extending Linear Relaxation for Non-Square Matrices and Soft Constraints

Noreen Jamil[1], Johannes Müller[1], M. Asif Naeem[2], Christof Lutteroth[3], Gerald Weber[3]

[1,3] *Department of Computer Science, The University of Auckland*
38 Princes Street, Auckland 1020, New Zealand.

[2] *School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology*
Private Bag 92006, Auckland, New Zealand.

Abstract

Linear relaxation is a common method for solving linear problems, as they occur in science and engineering. In contrast to direct methods such as Gauss-elimination or QR-factorization, linear relaxation is particularly efficient for problems with sparse matrices, as they appear in constraint-based user interface (UI) layout specifications. However, the linear relaxation method as described in the literature has its limitations: it works only with square matrices and does not support soft constraints, which makes it inapplicable to the UI layout problem.

To overcome these limitations we propose two algorithms for selecting the pivot elements used during linear relaxation: random pivot assignment, and a more complex deterministic pivot assignment. Furthermore, we propose three algorithms for solving specifications containing soft constraints: prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints. With these algorithms, it is possible to prioritize constraints: if there are conflicting constraints in a specification, as it is commonly the case for UI layout, only the constraints with lower priority are violated to resolve the conflicts.

[1] {njam031, jmue933}@aucklanduni.ac.nz
[2] mnaeem@aut.ac.nz
[3] {lutteroth, gerald}@cs.auckland.ac.nz

The performance and convergence of the proposed algorithms are evaluated empirically using randomly generated UI layout specifications of various sizes. The results show that our best linear relaxation algorithm performs significantly better than Matlab's LINPROG, a well-known efficient linear programming solver.

Keywords: UI layout, linear relaxation, soft constraints, non-square matrices

---

1. Introduction

Linear problems are encountered in a variety of fields such as engineering, mathematics and computer science. Hence, various numerical methods have been introduced to solve them. These methods can be divided into direct and indirect, also known as iterative, methods. Direct methods aim to calculate an exact solution in a finite number of operations, whereas iterative methods begin with an initial approximation and usually produce improved approximations in a theoretically infinite sequence whose limit is the exact solution [1].

Many linear problems are sparse, i.e. most linear coefficients in the corresponding coefficient matrix are zero so that the number of non-zero coefficients is $O(n)$ with n being the number of variables [2]. Since it is useful to have efficient solving methods specifically for sparse linear systems, much attention has been paid to iterative methods, which are preferable for such cases [3]. Iterative methods do not spend processing time on coefficients that are zero. Direct methods, on the other hand, usually lead to fill-in, i.e. coefficients change from an initial zero to a non-zero value during the execution of the algorithm. In these methods we therefore lose the sparsity property and have to deal with a lot more coefficients, which makes processing slower. Although there are some techniques to minimize fill-in effects, iterative, indirect methods are often faster than direct methods for large sparse problems [4].

One domain where sparse problems frequently occur is user interface (UI) layout. Section 2 describes the common properties of this domain, and delineates the solving approaches that have been proposed for it. The contributions of this

paper are motivated by and were evaluated for the UI layout problem.

One of the most common iterative methods used to solve sparse linear systems is linear relaxation. Starting with an initial guess, it repeatedly iterates through the constraints of a linear specification, refining the solution until a sufficient precision is reached. For each constraint, it chooses a *pivot variable* and changes the value of that variable so that the constraint is satisfied. Despite its efficiency for sparse systems, linear relaxation is currently not used for UI layout, for the reasons explained in the following.

A common property of many linear problems including UI layout is that the matrices corresponding to these linear problems are non-square. For example, when specifying UI layout with linear constraints, there are generally more constraints than variables. This is a problem for the common linear relaxation method, which assumes that the problem matrix is square and has a non-zero diagonal.

The problem for non-square matrices is that of *pivot assignment*, i.e. of choosing a pivot variable for each constraint during solving. The standard linear relaxation algorithms choose the pivot variable on the diagonal of the coefficient matrix. In case of square matrices with non-zero diagonals, this is an easy way to ensure that every constraint has a pivot variable, and that every variable is chosen once so that its value can be approximated. However, in the general case the diagonal approach has several problems:

1. Not every constraint contains an element on the diagonal of the problem matrix if there are more constraints than variables.
2. Diagonal elements may be zero, making them infeasible as pivot elements.
3. Diagonal elements may be small compared to the other elements on the same row of the matrix, making them a bad choice that may cause the solving process to diverge.

The common linear relaxation algorithms usually assume that a pivot assignment has been performed and that the chosen pivot elements are placed on the diagonal of the problem matrix. In square matrices this can always be

achieved by simple matrix transformations. However, in the case of non-square matrices, the problem number 1 above cannot be mitigated this way.

As a first contribution, we describe how the linear relaxation method can be extended to deal with the abovementioned problems. We propose two pivot assignment algorithms that can be used with any problem matrix, regardless of its shape or diagonal elements. The first algorithm selects pivot elements pseudo-randomly. The second algorithm selects pivot elements deterministically by optimizing certain selection criteria. The problem of pivot assignment in the case of non-square matrices and the two algorithms are explained in detail in Section 4.

Besides its inability to deal with non-square matrices, the common linear relaxation method has another shortcoming. Many problems, such as UI layout, may contain conflicting constraints. This may happen by over-constraining, i.e. by adding too many constraints, making a problem infeasible. If a specification contains conflicting constraints, the common linear relaxation method simply will not converge.

To deal with conflicts, *soft constraints* can be introduced. In contrast to the usual *hard* constraints, which cannot be violated, soft constraints may be violated if no other solution can be found. Soft constraints can be prioritized so that in a conflict between two soft constraints only the soft constraint with the lower priority is violated. This leads naturally to the notion of *constraint hierarchies*, where all constraints are essentially soft constraints, and the constraints that are considered "hard" simply have the highest priorities [5]. Using only soft constraints has the advantage that a problem is always solvable, which cannot be guaranteed if hard constraints are used.

In Section 5 we propose three conflict resolution algorithms for solving systems of prioritized linear constraints with the linear relaxation method. The first algorithm successively adds non-conflicting constraints in descending order of priority. The second algorithm starts with all constraints and successively removes conflicting constraints in ascending order of priority. The third algorithm is a mixture of both and adapts the binary search algorithm to the problem of

searching the best conflict-free subproblem. These algorithms yield conflict-free sub-problems to a given problem. There are some existing algorithms for finding feasible subproblems for sets of constraints [6]. However, they do not take into account prioritization of constraints, which is important for UI layout problems. There are a number of UI approaches to taking into account soft constraints [7, 8, 9, 10, 11].

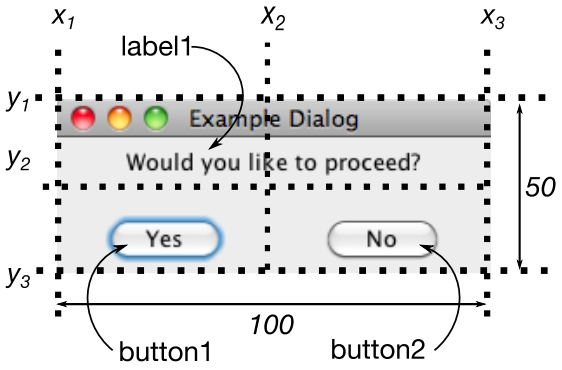
With the presented pivot assignment algorithms and the conflict resolution algorithms, linear relaxation can be applied to more linear constraint problems, such as constraint-based UI layout. Our two pivot assignment algorithms together with the three conflict resolution algorithms give a total of six different solution procedures. They were experimentally evaluated with regard to their convergence and performance, using randomly generated UI layout specifications. The results show that most of the proposed algorithms are feasible and efficient. Furthermore, we observed that some of our implemented solvers outperform Matlab's LINPROG linear optimization package [12], LP-Solve [13] and the implementation of QR-decomposition of the Apache Commons Math Library [14]. LP-Solve is a well-known linear programming solver that has been used for UI layout. The implementation of QR-decomposition of the Apache Commons Math Library is an example of a direct method. The methodology as well as the results of the evaluation can be found in Section 6.

2. Motivating Example for User Interface Layout

Constraints are a suitable mechanism for specifying the relationships among objects. They are used in the area of logic programming, artificial intelligence and UI specification. They can be used to describe problems that are difficult to solve, conveniently decoupling the description of the problems from their solution. Due to this property, constraints are a common way of specifying UI layouts, where the objects are widgets and the relationships between them are spatial relationships such as alignment and proportions. In addition to the relationships to other widgets, each widget has its own set of constraints

describing properties such as minimum, maximum and preferred size.

UI layouts are often specified with linear constraints [15]. The positions and sizes of the widgets in a layout translate to variables. Constraints about alignment and proportions translate to linear equations, and constraints about minimum and maximum sizes translate to linear inequalities. Furthermore, the resulting systems of linear constraints are sparse. There are constraints for each widget that relate each of its four boundaries to another part of the layout, or specify boundary values for the widget's size, as shown in Figure 1. As a result, the direct interaction between constraints is limited by the topology of a layout, resulting in sparsity.



$x_1 = 0, y_1 = 0, x_3 = 100, y_3 = 50$

label1 Hard: $x_3 - x_1 \geq 80, y_2 - y_1 \geq 20$
Soft: $x_3 - x_1 = x_3, y_2 - y_1 = y_3/2$

button1 Hard: $x_2 - x_1 \geq 40, y_2 - y_1 \geq 20$
Soft: $x_2 - x_1 = x_3/2, y_2 - y_1 = y_3/2$

button2 Hard: $x_3 - x_2 \geq 40, y_2 - y_1 \geq 20$
Soft: $x_3 - x_2 = x_3/2, y_2 - y_1 = y_3/2$

Figure 1: Example constraint-based UI layout with hard and soft constraints

For sparse linear problems, linear relaxation is known to perform very well. However, linear relaxation in its standard form cannot be applied to the UI layout problem for two reasons. First, the coefficient matrices are non-square: there are usually more constraints than variables. Second, many of the constraints are soft because they describe desirable properties in the layout (e.g. preferred sizes), which cannot be satisfied under all conditions (e.g. all layout

sizes). As a result, the existing UI layout solvers use algorithms other than linear relaxation. Some of these solvers will be discussed in Section 5.1.

3. Linear Relaxation

The approximate methods that provide solutions for systems of linear constraints starting from an initial estimate are known as iterative methods. Most of the research on iterative methods deals with iterative methods for solving linear systems of equalities and inequalities for sparse square matrices, the most important method being linear relaxation. This section summarizes the most important findings.

The best-known iterative method for solving linear constraints is the Gauss-Seidel method [1]. Given a system of n equations and n variables of the form

$$Ax = b \tag{1}$$

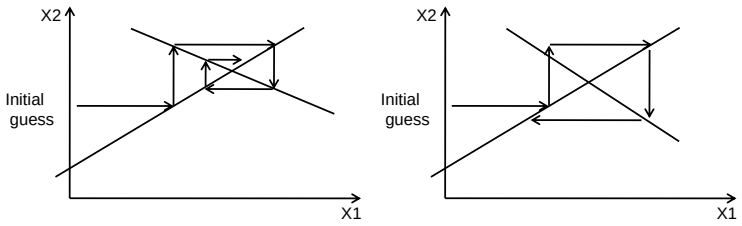
We can rewrite the equation for the ith term as follows

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j). \tag{2}$$

The variable x_i, which is brought onto the left side, is called the *pivot variable*, and a_{ii} is the *pivot coefficient* or *pivot element* chosen for row i. An initial estimate for x is chosen, which usually does not fulfill the equations. The algorithm refines the estimate by repeatedly replacing all individual x_i so that the ith equation becomes fulfilled. This is done in round-robin fashion, and one full run through all n equations is one iteration, r being the iteration number. We can therefore write the process as:

$$x_i^{r+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{r+1} - \sum_{j=i+1}^{n} a_{ij}x_j^r). \tag{3}$$

The algorithm iterates until the relative approximate error is less than a prespecified tolerance. Convergence and divergence behavior of Gauss-Seidel is shown in Figure 2.



(a) Convergence behavior (b) Divergence behaviour

Figure 2: Convergence and divergence behavior of Gauss-Seidel

Linear relaxation, also known as successive over-relaxation (SOR), is an improvement of the Gauss-Seidel method [16]. It is used to speed up the convergence of the Gauss-Seidel method by introducing a parameter ω, known as relaxation parameter, so that

$$x_i^{r+1} = \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{r+1} - \sum_{j=i+1}^{n} a_{ij}x_j^r) + (1-w)x_i^r. \tag{4}$$

This reduces to the Gauss-Seidel method if $\omega = 1$. It is known as over-relaxation if $\omega > 1$, and known as under-relaxation if $\omega < 1$.

3.1. Convergence

The convergence of the Gauss-Seidel method can be characterized in two related but quite distinct approaches. The first approach, which is the best-known theorem in this domain, is based on the coefficient matrix.

Definition 1. *A matrix is called strictly diagonally dominant if for all i*

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{5}$$

If a coefficient matrix is strictly diagonally dominant, the problem is guaranteed to converge [17]. However, this is only a sufficient condition, and a very strong condition that can be easily violated.

The notion of diagonal dominance gives naturally rise to a more general quantity that describes the influence of a variable.

Definition 2. *The influence of the kth variable in the ith constraint is*

$$\mathit{influence}_{ik} = \frac{|a_{ik}|}{\sum_j |a_{ij}|}. \tag{6}$$

All influences of variables in a constraint sum up to 1. If the constraints are normalized by dividing by the denominator above, then the absolutes of the coefficients of the variables are simply their influences. This metric will be helpful in defining our pivot assignment algorithms in Section 4.3.

The second approach to characterizing convergence is based on a derived matrix, the iteration matrix, and leads to a necessary and sufficient condition involving the spectral radius of the iteration matrix, defined as the absolute value of the maximum eigenvalue, which applies for SOR, not only Gauss-Seidel.

Definition 3 (Iteration Matrix)**.** *The changes to the estimate x^k in every step of the SOR method for $Ax = 0$ are given by a linear function. The matrix $M_\omega(A)$ of this function with $x^{k+1} = M_\omega(A)x^k$ is called the iteration matrix of $Ax = 0$.*

The SOR method converges for all initial values if and only if the spectral radius of the iteration matrix is smaller than one. If that condition is not fulfilled, the problem will diverge, except for some special initial values (such as the solutions itself). The smaller the spectral radius of the iteration matrix, the faster the SOR method converges.

An important characteristic of the SOR method with regard to convergence is translation invariance.

Lemma 1 (Translation Invariance)**.** *Let the SOR method for $Ax = b$ converge to $\bar{x}$ starting with x^0. Then the SOR method for $Ay = 0$ starting with $y^0 = x^0 - \bar{x}$ will have the same convergence behavior, i.e. for all j, r we have $y_j^r = x_j^r - \bar{x}_j$.*

The proof is by induction over j, r.

With Lemma 1 we can simplify the existing proof [18] of convergence based on the iteration matrix.

Theorem 1. *The SOR method for $Ax = b$ converges*[4] *if the Iteration Matrix is non-singular and $M(A)$ has a spectral radius smaller than* 1.

Proof. If the spectral radius is smaller than 1, then $M(A)^n$ converges to the matrix 0. Hence SOR for $Ax = 0$ converges to the correct solution 0. From translation invariance it follows that the SOR method for $Ax = b$ converges. □

Underrelaxation generally has better convergence behavior than Gauss-Seidel. But even for Gauss-Seidel, convergence is usually not a problem in practice. If there are concerns around convergence of the Gauss-Seidel method in a specific domain, preconditioners can be used. Good preconditioners for iterative methods are scaling algorithms [19] and bipartite matching algorithms [20]. These algorithms scale the infinity norm of both rows and columns in a matrix to 1 and permute large entries to the diagonal of a sparse matrix. We usually have well-conditioned coefficient matrices in the UI layout problem for which the Gauss-Seidel method converges quickly if appropriate pivot elements are chosen.

3.2. Inequalities

Linear relaxation supports linear equalities as well as inequalities. Inequalities are handled similarly to equalities [21, 22, 23]: in each iteration, inequalities are ignored if they are satisfied, and otherwise treated as if they were equalities. However, there are potential practical problems, which are described below using the following definitions.

Definition 4. *A system containing equalities as well as inequalities is called a mixed system.*

Definition 5. *The subsystem that consists of all the equations in a system of linear constraints is called maximum equality subsystem.*

[4] We treat inequalities as equalities or ignore them if they are fulfilled (cf. Section 3.2).

A mixed system with a square coefficient matrix cannot have a unique solution because this is only possible if there is an equality for each variable. In a typical mixed system, as it occurs for instance in UI layout specifications, the maximum equality subsystem is under-determined, i.e. there are fewer equalities than variables, and the whole system has more constraints than variables. This means that for typical mixed systems the standard linear relaxation algorithm, which works only on a square matrix, is insufficient. To use linear relaxation for such mixed systems we have to extend it, e.g. by applying the algorithms proposed in this article.

3.3. Advantages

Iterative methods such as linear relaxation have certain advantages over direct methods. Iterative methods are typically simpler to implement, resulting in smaller programs. Furthermore, they have fewer round-off errors than direct methods [3]. They start with an approximate answer and improve its accuracy in each iteration, so that the algorithm can be terminated once a sufficient accuracy is achieved. Ill-conditioned matrices are a problem for iterative methods just as much as for direct methods. Compared to direct methods, iterative methods are very efficient for sparse matrices, i.e. matrices where the number of non-zero elements is a small fraction of the total number of elements in the matrix. They are faster than direct methods because zero-coefficients are ignored implicitly, whereas direct methods have to process the zero-coefficients explicitly [3]. This implies that iterative methods need not store zero-coefficients explicitly, which leads to less memory consumption than with direct methods [24]. Considering these advantages, it would be useful if the limitations of linear relaxation could be overcome.

4. Non-Square Matrices

As pointed out in Section 3.2, mixed systems usually have a non-square matrix with more constraints than variables. Furthermore, they may have zero-coefficients on the diagonal. In some cases, they may also have more variables

than constraints (under-determined). In all these cases, the standard linear relaxation algorithm cannot be applied. In this section, we briefly present some related work on constraint problems with non-square coefficient matrices and propose two pivot assignment algorithms. Both algorithms help to overcome the limitations of the standard linear relaxation algorithm.

4.1. Related Work

Linear systems with non-square matrices are typically solved using direct methods, such as the QR-factorization method [17]. The QR-factorization method is used to solve linear systems of equations. Several methods can be used to compute QR-factorization, e.g. the Gram-Schmidt process, Householder transformations, or Givens rotations [17]. These methods require the calculation of a significant number of matrix norms, which slows them down as compared to other methods such as the normal equations method.

The normal equations method [25], which is also a direct method, is used to solve linear systems of the form $A^T Ax = A^T b$. It is fairly simple to program, but suffers from numerical instability when solving ill-conditioned problems. The condition of the normal equation matrix $A^T A$ is worse than that of the original matrix A. When an original matrix is converted into a normal equation matrix and a right hand-side vector, information can be lost. The normal equations method is considered the most common method despite the loss of information because it has been shown that its accuracy can be improved if iterative refinement is applied [26]. Some iterative methods like Gauss-Seidel use normal equations to solve non-square linear systems of equations [17].

There are some iterative methods [27] that can be used to solve systems of linear equations that are non-square. These methods include the simplex, the conjugate gradient and the generalized minimal residual methods. They have some limitations that make them inapplicable for some problems. These limitations are described as follows.

The simplex algorithm [28] is a well-known method used to solve linear programming problems. It is an iterative method, but one linear solving step

per iteration is required, which means this method cannot be faster than linear solving alone. It moves from one feasible corner point to another and continues iteration until an optimal solution is reached. The revised simplex method [29] is a variant of the simplex algorithm which is computationally more efficient for large sparse problems.

While the simplex algorithm tries to find an optimal solution according to an arbitrary linear objective function, there are other methods that try to find a least squares solution to an over-determined system of linear equations. These methods find a solution whose sum of squared errors is minimal. UI methods that work on squared errors exist such as [10] but also methods using absolute value [7] which is currently the basis for all layout in Apple,s Cocoa layout engine.

The generalized minimal residual method [30] is considered the most efficient method for solving least squares problems. One of the shortcomings of this method is its instability and poor accuracy of the computed solution due to the possibly high ill-conditioning of the normal equations system. This method is unstable because a non-square matrix is converted into a square matrix by applying normal equations.

The conjugate gradient method [31] can solve linear systems of equations if the matrix is symmetric and positive definite. However, it only works for well-conditioned problems as it cannot converge otherwise. Several methods for iteratively solving linear least squares problems – so called Krylov subspace methods – are surveyed in [32].

The Jacobi and Gauss-Seidel algorithms [33] are extended to solve non-square matrices in least squares sense by applying a hierarchical identification principle and by introducing block matrix inner-products. Numerical difficulties encountered for under-determined problems are the same in over-determined problems as described above. However, round-off errors accumulated in the under-determined case are more complicated than in the over-determined case because the solution is not unique.

4.2. Pivot Assignment

Since the diagonal elements do not lend themselves naturally as pivot elements if the matrix is non-square, we need to explicitly select a pivot element for each constraint. In other words, we need to determine a *pivot assignment*. Pivot assignment is also important for square matrices as it has an effect on convergence.

Definition 6. *A pivot assignment is an assignment of constraints to variables*

$$\gamma\colon \mathit{Constraints} \to \mathit{Variables}.$$

A *feasible* pivot assignment γ must be *surjective* and *total*. Surjectiveness is necessary because we require one constraint for each variable, otherwise the variable's value would not be changed by the algorithm. Totality is inherent in the definition of the linear relaxation algorithm, which requires a pivot variable for every constraint.

4.3. Pivot Assignment Algorithms

In the following we propose two pivot assignment algorithms, a random and a deterministic one. While the random algorithm avoids the issues of surjectiveness and totality by randomization, the deterministic algorithm ensures these properties, using the notions of most influential variables and constraints defined as follows.

Definition 7. *The most influential variable of a constraint is the one with the highest influence. The most influential constraint of a variable is the constraint where the variable has the highest influence.*

4.3.1. Random Pivot Assignment

The algorithm for random pivot assignment is depicted in Algorithm 1. The random algorithm assigns the pivot variable for each constraint randomly in each iteration (line 2). This means that in general the pivot assignment is changed in each iteration.

It is not inherently obvious that randomized assignments work for the linear relaxation approach, but it is the simplest approach that may work. Although the random algorithm does generally not make the optimal assignment with regard to convergence, it reduces the effect of bad assignments while allowing for good assignments. In particular, it is guaranteed that every suitable variable will be chosen as pivot variable at some point. The general assumption underlying randomized algorithms is that the effect of good choices outweighs the effect of bad choices.

Input: Constraints ($\mathbf{C}$)

Output: Pivot Assignment γ

1: **for** each constraint c **do**
2: Choose variable x of c randomly
3: Assign $\gamma(c) = x$
4: **end for**

Algorithm 1: Random pivot assignment

One of the drawbacks of random assignment is that it causes fluctuation of the error. This makes it harder to recognize whether the algorithm diverges, or whether fluctuations are only temporary. To address this problem, we propose a deterministic approach in the following section.

4.3.2. Deterministic Pivot Assignment

The algorithm for deterministic pivot assignment is presented in Algorithm 2. It creates a single pivot assignment that is used consistently during the solving process. It is explained in the course of the following proof of correctness.

Theorem 2. *The deterministic algorithm produces only feasible assignments.*

Proof. In lines 1–7 each constraint is assigned a variable x, therefore the resulting assignment is total. In lines 8–11 every variable y that has not been assigned a constraint yet is assigned a new constraint, which is a duplicate of an existing constraint. As a result, the resulting assignment is surjective.

If the matrix is diagonally dominant, at the time the algorithm iterates over a particular constraint, the most influential variable of this constraint will still be unassigned. After the first loop, there will be no unassigned variables left. As a result, the algorithm chooses the diagonal elements as pivots in the case of diagonally dominant matrices. Duplicating constraints of a problem does not change the problem, hence this is a valid transformation. □

Input: Constraints ($\mathbf{C}$)

Output: Pivot Assignment γ

1: **for** each constraint c **do**
2: **if** some variables of c are still unassigned **then**
3: Choose unassigned variable x of c with the largest influence, assign $\gamma(c) = x$
4: **else**
5: Choose the most influential variable x of c, assign $\gamma(c) = x$
6: **end if**
7: **end for**
8: **for** each still unassigned variable y **do**
9: Find the most influential constraint c for y
10: Duplicate c to c', assign $\gamma(c') = y$
11: **end for**

Algorithm 2: Deterministic pivot assignment

In the general case constraint-based UIs are over-determined, which can result in conflicts between constraints of the problem. A proper pivot assignment algorithm alone is not sufficient to deal with such cases. A technique to handle conflicts between constraints, e.g. in the form of soft constraints, is required.

5. Soft Constraints

Hard constraints are constraints that must always be satisfied. If this is impossible, there is no solution. For many problems, including UI layout, con-

flicting constraints occur naturally in specifications, as they express properties of a solution that are desirable but not mandatory. As a result, soft constraints need to be supported, which are satisfied if possible, but do not render the specification infeasible if they are not. A natural way to support soft constraints is to treat all constraints as soft constraints, with different priorities (p). These priorities can be defined as a total order on all constraints that specifies which one of two constraints should be violated in case of a conflict.

To define the solution of a system of prioritized soft constraints, we first have to define the subset $E \subseteq \mathit{Constraints}$ of *enabled constraints*. We consider the characteristic function $\mathbf{1}_E : \mathit{Constraints} \to \{0, 1\}$ of E, which expresses whether a constraint is contained in E, to construct an integer in binary representation (ι). According to their priority, each constraint is represented by a bit of that integer, with constraints of higher priority taking the more significant bits. The value of the characteristic function for the constraint with the highest priority is considered the most significant bit. Then such subsets can be compared by using the numerical order $\geq$ of the integers. We are interested in the subset that is largest in that order while containing only non-conflicting constraints. In the following, we discuss existing approaches for solving linear soft constraints. Then, we describe three algorithms that address support for soft constraints in the linear relaxation method: prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints.

5.1. Related Work

All constraint solvers for UI layout must support over-determined systems. The commonly used techniques for dealing with over-determined problems are weighted constraints and constraint hierarchies [34, 35, 36]. Weighted constraints are typically used with some general forms of direct methods, while constraint hierarchies are especially utilized in linear programming based algorithms. Many UI layout solvers are based on linear programming and support soft constraints using slack variables in the objective function [7, 8, 9, 10, 15, 37].

Most of the direct methods for soft constraint problems are least squares

methods such as LU-decomposition and QR-decomposition [38]. The UI layout solver HiRise [39] is an example of this category. Its successor, HiRise2 [40] solves hierarchies of linear constraints by applying an LU-decomposition-based simplex method.

However, if weights are used to express a hierarchy of constraints the differences between them have to be very large. This in turn can push numerical limits. Hence, it is desirable to enforce priorities of constraints without using weights directly.

Many different local propagation algorithms have been proposed for solving constraint hierarchies in UI layouts. The DeltaBlue [41], SkyBlue [42] and Detail [43] algorithms are examples of this category. The DeltaBlue and SkyBlue algorithms cannot handle simultaneous constraints that depend on each other. However, the Detail algorithm can solve constraints simultaneously based on local propagation. All of the methods to handle soft constraints utilized in these solvers are designed to work with direct methods, so they inherit the problems direct methods usually have with sparse matrices. Some orthogonal projection methods are also extended to solve soft constraints for User Interface Layout problems [44].

QUICKXPLAIN [45] tries to find a conflict-free constraint system by successively adding or removing constraints from the group of constraints. The groups of constraints to be added or removed are determined by a recursive decomposition of the problem. It is mixture of QuickSort and MergeSort in the sense that QuickXplain must solve both of the two subproblems resulting from the problem decomposition. In contrast to QuickSort and MergeSort, the result of the right subproblem, which is solved first by QuickXplain, has an impact on the formulation of the left subproblem. This is a particularity of QuickXplain.

The Maximum Satisfiability (MAXSAT) problem is a generalization of Satisfiability (SAT) that can represent optimization problems. SAT problem tries to find an assignment that satisfies all the constraints if one exists otherwise no assignment exists which can be satisfiable. The goal of MAXSAT is to find an assignment that satisfies maximum number of constraints. The MAXSAT

solvers are based on branch and bound solvers and satisbility testing [46, 47, 48, 49, 50, 51, 52].

The problem of finding the largest possible subset of constraints that has a feasible solution given a set of linear constraints is widely known as the maximum feasible subsystem (MaxFS) problem [6]. The dual problem of MaxFS is the problem of finding the irreducible infeasible subsystem (IIS) [53]. If one more constraint is removed from an IIS, the subsystem will become feasible. For both problems different solving methods were proposed.

To solve the MaxFS problem, non-deterministic as well as deterministic methods were proposed. Some of these methods use heuristics [54, 55]. Only a few methods solve the problem deterministically. The branch and cut method proposed by Pfetsch[56] is such a deterministic method. Besides methods to solve MaxFS, there are some methods to solve the IIS problem. These methods are: deletion filtering, IIS detection algorithm and grouping constraints method.

Deletion filtering [57] starts with the set of all constraints. For each constraint in the set, the method temporarily drops the constraint from the set and checks the feasibility of the reduced set. If the reduced set is feasible, the method returns the dropped constraint to the set of constraints. If the reduced set is infeasible, the method removes the dropped constraint permanently. Baker et al. [58] proposed an algorithm, Diagnosis of Over-determined Constraint Satisfaction Problems. This algorithm tries to find the set of least important constraints that can be removed to solve the remaining constraint satisfaction problem. If the solution is not optimal then it tries to find next-best sets of least-important constraints until an optimal solution is found.

The IIS detection algorithm [59, 60, 61] starts with a single constraint and adds constraints successively. If the system is infeasible after adding a new constraint, then the method discards this new constraint.

The grouping constraints method was introduced by Guieu and Chinneck [62] to speed up the IIS detection algorithm and deletion filtering. It adds or drops constraints in groups by using the deletion filtering or IIS detection algorithms.

Even though these methods deal with the problem of finding a feasible sub-

system, it is not possible to apply them directly. The main reason is that they do not consider prioritized constraints, as necessary for problems such as UI layout. As discussed in Section 5, we have to find not only the set with the maximum number of constraints, but the set of constraints with $max(\iota)$. We call this problem *prioritized MaxFS*, and propose as a solution the following algorithms: prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints.

5.2. Prioritized IIS Detection

In prioritized IIS detection, which is depicted as Algorithm 3, we start with an empty set E of enabled constraints (line 1). We add constraints incrementally in order of descending priority so that E is conflict-free, until all non-conflicting constraints have been added. Iterating through the constraints, we add each constraint tentatively to E ("enabling" it), and try to solve the resulting specification (line 7). Note that whenever a constraint is added, the pivot assignment needs to be recalculated. If a solution is found, we proceed to the next constraint. If no solution is found, the tentatively added constraint is removed again. In that case, the previous solution is restored and we proceed to the next constraint.

This algorithm assumes that the method used for solving converges if there is no conflict. The algorithm approximates $max(\iota)$ starting from the most significant bit and progressing down to the least significant bit. This property distinguishes our algorithm from the existing IIS detection algorithm. The best case time complexity of this algorithm is $O(nlog(n))$ and the worst case is $O(n^2)$.

5.3. Prioritized Deletion Filtering

Prioritized deletion filtering is an algorithm that assumes that a predicate $conflicting(c)$ with certain properties exists. The assumption is that during one unsuccessful attempt to solve the current specification, we can collect reliable information on each single constraint whether it is conflicting.

Input: Constraints (**C**)

Output: Non-conflicting constraints

```
1: DISABLE(C)
2: SORT(C) (by priority)
3: for each constraint c in order of priority, descending do
4:    Remember current variable values
5:    ENABLE(c)
6:    Assign pivot elements for all constraints
7:    Apply linear relaxation
8:    if solution not found then
9:       DISABLE(c)
10:      Restore old variable values
11:   end if
12: end for
```

Algorithm 3: Prioritized IIS Detection

The steps are shown in Algorithm 4. We start with all constraints enabled, i.e. $E = \mathit{Constraints}$ (line 1). We try to solve the specification, and if a solution is found, this means E is conflict-free. In this case, we return the solution. Otherwise, we remove the conflicting constraint with the lowest priority from E ("disabling") and recalculate the pivot assignment.

With a very simple heuristic predicate based on the error fluctuation, as described below in detail, this algorithm was used quite successfully during our evaluation. However, even if one assumes a completely reliable predicate $conflicting(C)$, the set of constraints getting disabled is generally larger than allowed in our definition of soft constraints. In contrast to prioritized IIS detection, if there is a conflicting constraint c of a higher priority in a specification, that constraint will only be deleted after it might have already triggered removal of a constraint d of a lower priority. After c is removed, d might be solvable, but has already been lost. We present this approach to provide another perspective on addressing soft constraints, and as a motivation for our third algorithm

described in Section 5.4.

Currently, we use the following heuristic: $conflicting(c)$ is true if the value of its pivot variable $\gamma(c)$ has been changed significantly during the last linear relaxation iteration. While this condition is true for conflicting constraints, it is not a sufficient condition, as other non-conflicting constraints may be affected by a conflict and hence satisfy this condition, too. The best case time complexity of this algorithm is $O(nlog(n))$ and the worst case is $O(n^3)$.

Input: Constraints (**C**)

Output: Non-conflicting constraints

```
1: ENABLE(C)
2: SORT(C) (by priority)
3: for each constraint do
4:    Assign pivot elements for all constraints
5:    Apply linear relaxation
6:    if solution is found then
7:       return solution
8:    end if
9:    for each constraint c in order of priority, ascending do
10:      if conflicting(c) then
11:         DISABLE(c)
12:         break
13:      end if
14:   end for
15: end for
```

Algorithm 4: Prioritized Deletion Filtering

5.4. Prioritized Grouping Constraints

The prioritized grouping constraints algorithm is a combination of prioritized IIS detection and prioritized deletion filtering. It tries to find a conflict-free constraint system by successively adding or removing constraints from the

system of constraints. If a constraint system is conflict-free, the algorithm adds constraints; if it has conflicts, the algorithm removes constraints. It adds and removes not only one constraint at a time, but also groups of constraints, whereby the size of the groups follows the classic patterns of a binary search approach. The algorithm stops if the system is conflict-free and no more constraints can be added. In contrast to prioritized deletion filtering, this algorithm does not require a predicate $conflicting(c)$.

The steps are shown in Algorithm 5. First, the algorithm is initialized by sorting the list of constraints ($\mathbf{C}$) (line 1) and initializing some variables (line 2). The variables *beginning* and *end* determine the upper-inclusive and the lower-exclusive bounds of the area of the list of constraints which possibly contains conflicting constraints, the *s*earch window. These bounds are adjusted while the algorithm is running. When initializing the algorithm, we set $end = 2|\mathbf{C}|$, which is adjusted to $end = |\mathbf{C}|$ in the first iteration. In the second iteration we check the whole list from the first entry ($beginning = 0$) to the last entry ($end = |\mathbf{C}| - 1$).

After initialization the algorithm enters a loop which iteratively finds the prioritized MaxFS ($max(\iota)$). First, the algorithm checks whether a solution was found in the previous step is feasible. If this is the case, we know that, at least up to *end*, no conflict is in the constraint list. We can, therefore, set the upper bound *beginning* of the search window to *end*, ignore the old search window in the following iterations, and increase the lower bound *end* to form a new window. The variable value *end* is increased exponentially with δ.

If the solution is not found, we have either identified a conflicting constraint, or we still need to decrease the size of the search window exponentially (line 15). We have found a conflicting constraint if the size of the search window is shrunken to a single constraint (line 10). In that case, we deactivate this constraint (line 11), move the upper bound of our search space to index position *end* (line 12), and set the size of our search window to one (line 13) for the following iteration.

Now, after the bounds of the search window are calculated, the constraints

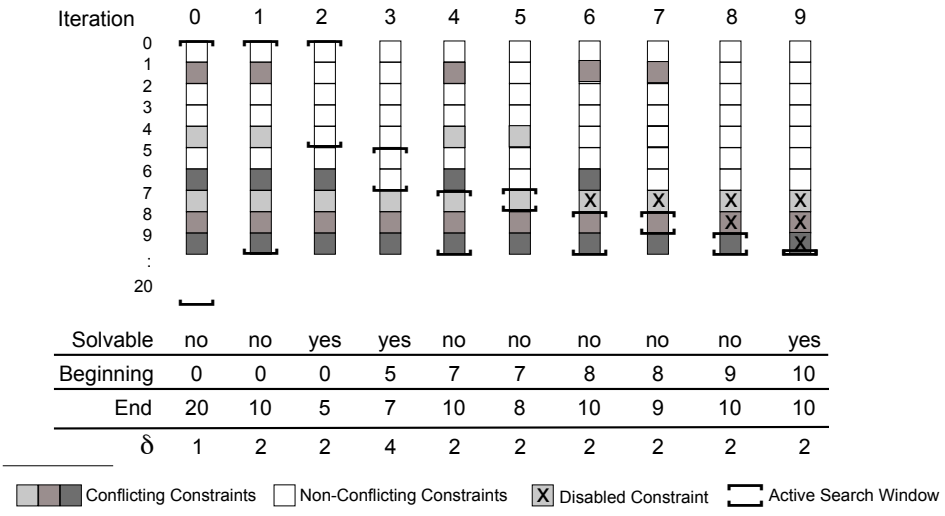


Figure 3: Example run of the prioritized grouping constraints algorithm

within the search window $(beginning, end)$ are enabled and all constraints below the search window are disabled (line 18). The constraints above *beginning* are still enabled from the preceding iterations. Finally, the pivot elements for the enabled constraints are determined and the problem is solved for the enabled constraints (lines 19 and 20).

Similar to the aforementioned algorithms, this algorithm assumes that linear relaxation converges if there is no conflict in the system. Under that assumption, it finds $\max(\iota)$, i.e. it will end with the same constraints as prioritized IIS detection. However, in contrast to prioritized IIS detection, it adds and removes constraints in groups, which reduces the number of required iterations in most cases. The best case time complexity of this algorithm is $O(log(n))$ and the worst case is $O(n)$.

Figure 3 depicts a run of the prioritized grouping constraints algorithm. In this example, the problem consists of 10 constraints, ordered according to their priority. The constraint pairs (2, 9), (5, 8) and (7, 10) are conflicting. The objective is to find a problem with $max(\iota)$.

The algorithm starts in iteration 0 by just initializing the size of the search

Input: Constraints ($\mathbf{C}$)

Output: Non-conflicting constraints

1: SORT($\mathbf{C}$) by priority
2: $\delta \leftarrow 1$; $beginning \leftarrow 0$; $end \leftarrow 2(|\mathbf{C}|)$
3: **while** $beginning < |\mathbf{C}|$ **do**
4: **if** solution found **then**
5: Remember current variable values
6: $beginning \leftarrow end$
7: $end \leftarrow end + \delta$ (or $|\mathbf{C}|$ if it is out of bounds); $\delta \leftarrow 2\delta$
8: **else**
9: Restore old variable values
10: **if** $end = beginning + 1$ **then**
11: DISABLE($\mathbf{C}[beginning]$)
12: $beginning \leftarrow end$
13: $end \leftarrow end + 1$ (or $|\mathbf{C}|$ if it is out of bounds); $\delta \leftarrow 2$
14: **else**
15: $end \leftarrow beginning + \frac{end - beginning}{2}$
16: **end if**
17: **end if**
18: ENABLE($\mathbf{C}[beginning \ldots (end - 1)]$) and DISABLE($\mathbf{C}[end \ldots (|| - 1)]$)
19: Assign pivot elements for all enabled constraints
20: SOLVE enabled constraints
21: **end while**

Algorithm 5: Prioritized grouping constraints

window and solving the complete list for the first time. With that result it enters iteration 1 with a search window that consists of the complete list of constraints (0 – 9), and tries to solve them. This problem cannot be solved since we have three conflicts in the list, as described above. In iteration 2, the search window is halved and the algorithm tries to solve the problem in the upper part (0 – 4). This subproblem is solvable and the algorithm moves the beginning of the

search window to index 5 and starts with a new search window of size $\delta = 2$ in iteration 3. The new subproblem is solvable as well and δ is doubled to 4, so that the subproblem in iteration 4 contains again the whole list of constraints which are not solvable. In iteration 5 the search window is halved again. Since only one constraint is in the search window left and the problem is still not solvable, the problem in the search window must be conflicting with one of the higher prioritized constraints and has to be disabled. The new search window in iteration 6 starts below the disabled constraint with size 2. Again it contains conflicting constraints and is halved, which yields a search window of size 1 in the next iteration and a subproblem which is still not solvable. Hence constraint 8 must be conflicting as well and is disabled. The search window is moved below the disabled constraint in iteration 8. The new subproblem is not solvable and constraint 9 is disabled. In the last iteration, the search window is of size 0 and the problem is solvable, which indicates that all conflicts are found and the algorithm can terminate. As the example shows, the prioritized grouping constraints algorithm deactivates only lower prioritized conflicting constraints, resulting in $max(\iota)$. This property distinguishes our algorithm from the existing grouping constraints algorithm.

6. Experimental Evaluation

In this section, we present an experimental evaluation of the proposed algorithms. We conducted two different experiments to evaluate (1) the convergence behavior, (2) the performance in terms of computation time.

6.1. Methodology

For both experiments we used the same computer and test data generator, but instrumentalized the algorithms differently. We used the following setup: a desktop computer with Intel i5 3.3GHz processor and 64-bit Windows 7 running an Oracle Java virtual machine. Layout specifications were randomly generated using the test data generator described in [15]. For each experiment the same

set of test data was used. The specification size was varied from 4 to 2402 constraints, in increments of 4 (2 new constraints for the position and 2 new constraints for the preferred size of a new widget). For each size, 10 different layouts were generated, resulting in a total of 6000 different layout specifications that were evaluated. A linear relaxation parameter of 0.7 and a tolerance of 0.01 were used for linear relaxation. We use 1000 maximum number of iterations until the algorithm gets near optimal solutions or an indication of likely infeasibility of the system.

In Experiment 1, we investigated the convergence behavior of each algorithm by measuring the number of sub-optimal solutions. A solution is sub-optimal if the error of a constraint (the difference between right hand and left hand side) is not smaller than the tolerance.

In Experiment 2, we measured the performance in terms of computational time T in milliseconds (ms), depending on the problem size measured in number of constraints c. Each of the proposed algorithms was used to solve each of the problems of the test data set, and the time was taken. As a reference, all the generated specifications were also solved with Matlab's LINPROG solver [12] and LP-Solve [13]. We selected these two solvers because LINPROG is widely known for its speed [5], and LP-Solve was previously used to solve UI layout problems [15]. Additionally, we wanted to know how well our algorithms compete with a direct method. Hence we also used the implementation of QR-decomposition of the Apache Commons Mathematics Library [14], which is a freely available open-source library.

Algorithm 6 shows how random sets of areas A are created by partitioning the bounding area of a GUI. First, A only contains the bounding area of the GUI. Then, while the number of areas $|A|$ in A is less than the number of areas n_{areas} that the layout should contain, we divide one of the existing areas, thus increasing the total number of areas by one. Line 4 randomly chooses an area of A, and line 5 removes it from A. Then, we randomly decide whether to

[5] http://plato.asu.edu/ftp/lpfree.html

1: **function** GENERATE(n_{areas})
2: A $\leftarrow$ $(left, top, right, bottom)$
3: **while** $|A| < n_{areas}$ **do**
4: a $\leftarrow randomElement(A)$
5: $A \leftarrow A - (a)$
6: **if** $random < 0.5$ **then**
7: $A \leftarrow AU(a.left, a.top, x_{new}, a.bottom), (x_{new}, a.top, a.right, a.bottom)$
8: **else**
9: $A \leftarrow AU(a.left, a.top, a.right, y_{new}), (x.left, y_{new}, a.right, a.bottom)$
10: **end if**
11: **end while**
12: **end function**

Algorithm 6: Generation of a random partition of areas.

divide a vertically or horizontally. Random is a random value between 0 and 1. x_{new} is a new x-tab that is inserted in order to divide a vertically; y_{new} is a new y-tab that is inserted in order to divide a horizontally. Because of the growing number of smaller areas and the uniformly random choice of the area that is subdivided next, the algorithm produces layouts with some large areas and many small ones. In our tests, we put a button into each of the areas of the generated layouts. We chose the GUI size randomly between (100 100, 800 600). A random minimum size was set for each area to be (10 $(400/n_{areas})$, 10 $(300/n_{areas})$. Minimum sizes are important for the controls in the areas to be rendered correctly, and it is important to reduce the maximum for each minimum size with increasing *nareas* in order to have a feasible layout. If the minimum sizes are too large then the minimum widths or minimum heights of adjacent areas might add up to more than the total size of the GUI so that there cannot be a solution. For each area, p_{expand} and p_{shrink} are chosen randomly between 0 and 1.

6.2. Results

In Experiment 1, we found that all algorithms converge. This result is obvious since the algorithms are designed to find a solvable subproblem.

In Experiment 2, we analyzed the trends of the computational performance of the algorithms using different regression models (linear, quadratic, cubic and log). We found that the best-fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon.$$

Key parameters of the models are depicted in Table 2; a graphical representation of the models can be found in Figures 4 – 6. Table 1 explains the symbols used.

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination of the estimated regression models

Table 1: Symbols used for the performance regression model

For some strategies, some parameters do not have a significant effect. That can be interpreted as the complexity of the algorithm not following a certain polynomial trend. For example, prioritized deletion filtering with deterministic pivot assignment seems to have a purely quadratic runtime behavior. For a better comparison of the runtime behavior of the strategies, we considered all combinations of soft constraint and pivot selection algorithms. Figure 4 illustrates the performance comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints using random pivot assignment. As the graphs indicate, prioritized grouping constraints exhibit better performance than prioritized deletion filtering and prioritized IIS detection.

Figure 5 compares prioritized IIS detection, prioritized deletion filtering and

Strategy	β_0	β_1	β_2	β_3	R^2
Pr. grouping constraints / deter.	$1.283e^{+03***}$	$-1.045e^{+01***}$	$2.350e{-}02^{***}$	$5.865e^{-06***}$	0.9877
Pr. grouping constraints / random	$8.341e^{+00***}$	$-4.692e^{-02***}$	$1.225e^{-04***}$	$-1.305e^{-08***}$	0.9917
Pr. deletion filtering / random	$-6.399e^{-01}$	$5.333e^{-03}$	$8.946e^{-05***}$	$2.831e^{-09***}$	0.9925
Pr. IIS detection / random	$4.174e^{+00***}$	$-2.270e^{-02***}$	$1.620e^{-04***}$	$-1.087e^{-08***}$	0.9957
Pr. IIS detection / deter.	$-6.195e^{+02***}$	$3.953e^{+00***}$	$-4.368e^{-03***}$	$1.243e^{-05***}$	0.9971
Pr. deletion filtering / deter.	$1.537e^{+00}$	$-7.698e^{-03}$	$1.668e^{-04***}$	$7.015e^{-10}$	0.9893
LINPROG	$1.829e^{+01***}$	$1.591e^{-04}$	$4.934e^{-05***}$	$1.577e^{-08***}$	0.9367
LP-Solve	$-2.491e^{+00***}$	$3.924e^{-02***}$	$2.079e^{-04***}$	$1.904e^{-08***}$	0.9900
QR-Decomposition	$-3.770e^{+01***}$	$2.802e^{-01***}$	$-4.009e^{-04***}$	$2.850e^{-07***}$	0.9989

Significance codes: *** $p < 0.001$

Table 2: Regression models for the different solving strategies

prioritized grouping constraints using deterministic pivot assignment. Generally, these strategies are slower than the strategies with random pivot assignment because the computation of the pivot assignment is more complex and takes longer. The slowest strategy is prioritized grouping constraints with deterministic pivot assignment, followed by prioritized IIS detection with deterministic pivot assignment. Prioritized deletion filtering with deterministic pivot assignment has the best performance. The runtime of prioritized deletion filtering with deterministic pivot assignment appears almost linear in the number of constraints. This is due to the fact that for prioritized deletion filtering, the pivot assignment needs only be recomputed for each conflicting constraint. The runtime performance of prioritized grouping constraints has the highest volatility. This is due to the fact that the performance of prioritized grouping constraints depends on the distribution of conflicting constraints over the list of constraints. If conflicting constraints are close, the algorithm searches only a small fraction of the whole list. If conflicting constraints are almost equally distributed over the list of constraints, the algorithm searches the whole list.

Figure 6 compares all the aforementioned algorithms, except for the slow

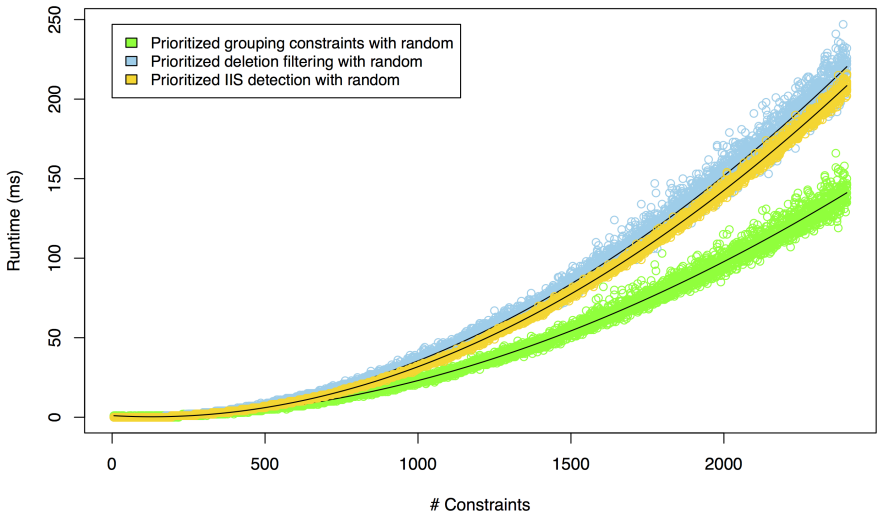


Figure 4: Performance comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints using random pivot assignment

prioritized IIS detection and prioritized grouping constraints with deterministic pivot assignment, to LINPROG, LP-Solve and QR-decomposition. Generally, all our algorithms perform significantly better than LINPROG, LP-Solve and QR-decomposition, especially for bigger problems, with prioritized grouping constraints with random pivot assignment exhibiting the best runtime behavior.

References

[1] A. B. Saeed, A. B. Naeem, Numerical Analysis, Shahryar, 2008.

[2] S. Kunis, H. Rauhut, Random sampling of sparse trigonometric polynomials, ii. orthogonal matching pursuit versus basis pursuit, Journal Foundations of Computational Mathematics 8 (6) (2008) 737–763. doi:10.1007/s10208-007-9005-x.

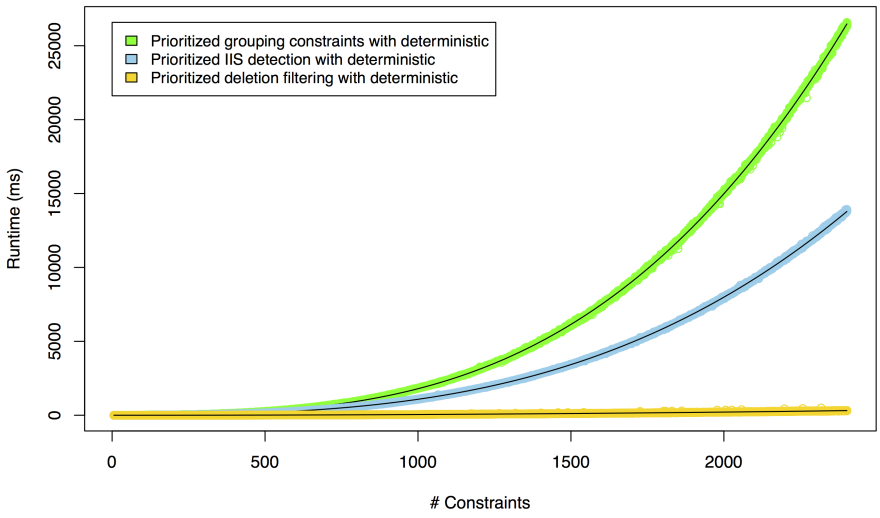


Figure 5: Runtime comparison of prioritized IIS detection, prioritized deletion filtering and prioritized grouping constraints with deterministic pivot assignment

[3] H. M. Anita, Numerical Methods for Scientist and Engineers, Birkhauser, 2002.

[4] M. Benzi, Preconditioning techniques for large linear systems: A survey, Journal of Computational Physics 182 (2002) 418–477.

[5] A. Borning, B. Freeman-Benson, M. Wilson, Constraint hierarchies, Lisp and Symbolic Computation 5 (3) (1992) 223–270.

[6] J. W. Chinneck, Fast heuristics for the maximum feasible subsystem problem, Informs Journal of Computation (2001) 210–223.

[7] G. J. Badros, A. Borning, P. J. Stuckey, The cassowary linear arithmetic constraint solving algorithm, ACM Transactions on Computer-Human Interaction 8 (4) (2001) 267–306.

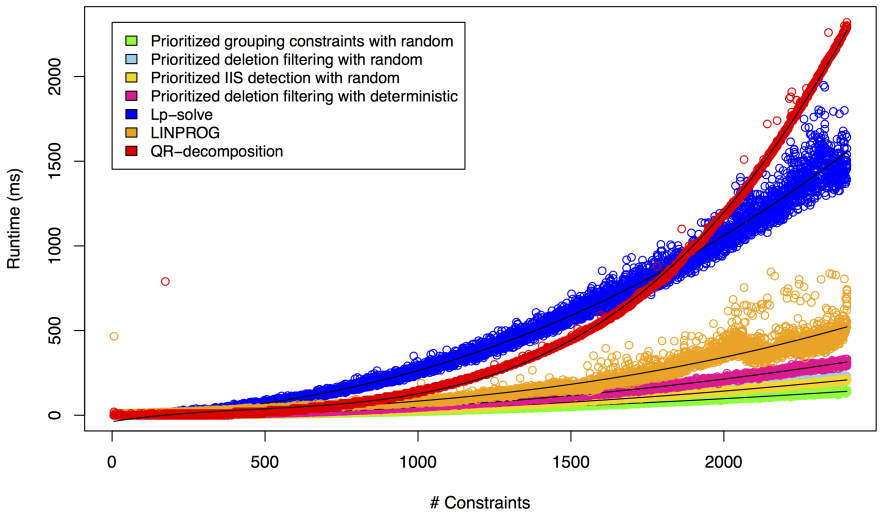


Figure 6: Performance comparison of the best solving strategies with LINPROG, LP-Solve and QR-decomposition

[8] A. Borning, K. Marriott, P. Stuckey, Y. Xiao, Solving linear arithmetic constraints for user interface applications, in: Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST '97), ACM, 1997, pp. 87–96. doi:10.1145/263407.263518.
URL http://doi.acm.org/10.1145/263407.263518

[9] M. S. Bazaraa, J. J. Jarvis, H. D. Sherali, Linear Programming and Network Flows, 1990.

[10] K. Marriott, S. C. Chok, A. Finlay, A tableau based constraint solving toolkit for interactive graphical applications, in: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming, CP '98, Springer, London, UK, 1998, pp. 340–354.

[11] N. Jamil, X. Chen, A. Cloninger, Randomized hildreth's algorithm with

applications to soft constraints for user interface layout, Journal of Computational and Applied Mathematics 288 (2015) 193–202.

[12] J. Stuart, Linprog:http://www.mathworks.com/matlabcentral/fileexchange/97-linprog/.

[13] M. Berkelaar, P. Notebaert, K. Eikland, A (mixed integer) linear programming problem solver: `http://lpsolve.sourceforge.net/`.

[14] Apache Software Foundation, Commons math, release 2.1; `http://commons.apache.org/math` (2012).

[15] C. Lutteroth, R. Strandh, G. Weber, Domain specific high-level constraints for user interface layout, Constraints 13 (3).

[16] D. M. Young, Jr, Iterative Solution of Large Linear Systems, Academic Press, 1971.

[17] B. N. Datta, Numerical Linear Algebra And Applications, Cole Publishing, 1995.

[18] Y. Saad, Iterative methods for sparse linear systems, SIAM, Philadelphia, PA (2003) 112.

[19] D. Ruiz, A scaling algorithm to equilibrate both rows and columns norms in matrices, Tech. rep. (2001).

[20] I. S. Duff, J. Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, Tech. rep. (1999).

[21] J. L. Goffin, The relaxation method for solving systems of linear inequalities, Mathematics of Operations Research 5 (3) (1980) 388–414.

[22] S. Agmon, The relaxation method for linear inequalities, Canadian Journal of Mathematics (1954) 382–392.

[23] T. Motzkin, I. Schoenberg, The relaxation method for linear inequalities, Canadian Journal of Mathematics (1954) 393–404.

[24] R. L. Burden, J. Faires, Numerical Analysis, Bob Pirtle, 2005.

[25] M. T. Heath, Scientific Computing, An Introductory Survey, The McGraw-Hill Companies, 1997.

[26] L. V. Foster, Modifications of the normal equations method that are numerically stable, In Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms (1991) 501–512.

[27] O. Axelsson, Iterative Solution Methods, Cambridge Uni. Press, 1996.

[28] G. B. Dantzig, Linear Programming and Extensions, 11th Edition, Princeton Landmarks in Mathematics, Princeton Uni. Press, Princeton NJ, USA, 1998.

[29] H. A. Taha, Operations Research: An Introduction, Mcmillan Publishing, 1992.

[30] Y. Saad, M. H. Schultz, Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 7 (3) (1986) 856–869. doi:10.1137/0907058.
URL http://dx.doi.org/10.1137/0907058

[31] M. R. Hestenes, E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems, Journal of Research of the National Bureau of Standards 49 (1952) 409–436.

[32] G. Golub, C. Van Loan, Matrix Computations, Johns Hopkins Uni. Press, 1996.

[33] F. Ding, T. Chen, Iterative least-squares solutions of coupled sylvester matrix equations, Systems and Control Letters 54 (2005) 95–107.

[34] P. Meseguer, N. Bouhmala, T. Bouzoubaa, M. Irgens, M. Sánchez, Current approaches for solving over-constrained problems, Constraints 8 (1) (2003) 9–39.

[35] H. Hosobe, S. Matsuoka, A foundation of solution methods for constraint hierarchies, Constraints 8 (1) (2003) 41–59.

[36] E. C. Freuder, Partial constraint satisfaction, International Joint Conference on Artificial Intelligence (1989) 278–283.

[37] C. Zeidler, C. Lutteroth, G. Weber, Constraint solving for beautiful user interfaces: How solving strategies support layout aesthetics, In Proceedings of CHINZ (2012) 23–32.

[38] Y. Yoshioka, H. Masuda, Y. Furukawa, A constrained least squares approach to interactive mesh deformation, in: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006, SMI '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 23–33. doi:10.1109/SMI.2006.1.
URL http://dx.doi.org/10.1109/SMI.2006.1

[39] H. Hosobe, A scalable linear constraint solver for user interface construction, in: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP '02, Springer, London, UK, 2000, pp. 218–232.

[40] H. Hosobe, A simplex-based scalable linear constraint solver for user interface applications, in: Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on, 2011, pp. 793–798. doi:10.1109/ICTAI.2011.124.

[41] J. M. Freeman-Benson, A. Borning, An incremental constraint solver, Communications of the ACM 33 (1) (1990) 54–63.

[42] M. Sannella, Skyblue: a multi-way local propagation constraint solver for user interface construction, in: Proceedings of the 7th annual ACM symposium on User interface software and technology (UIST '94), ACM, 1994, pp. 137–146. doi:10.1145/192426.192485.
URL http://doi.acm.org/10.1145/192426.192485

[43] H. Hosobe, K. Miyashita, S. Takahashi, S. Matsuoka, A. Yonezawa, Locally simultaneous constraint satisfaction, in: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP '94, Springer, London, UK, 1994, pp. 51–62.
URL http://dl.acm.org/citation.cfm?id=645814.668743

[44] N. Jamil, J. Müller, D. Needell, C. Lutteroth, G. Weber, Kaczmarz algorithm with soft constraints for user interface layout, In Proceedings of 25th International Conference on Tools with Artificial Intelligence (ICTAI) (2013) 818–824.

[45] U. Junker, Quickxplain: preferred explanations and relaxations for over-constrained problems, Proceedings of the 19th national conference on Artifical intelligence (2004) 167–172.

[46] C. M. Li, F. Many, N. O. Mohamedou, J. Planes, Exploiting cycle structures in max-sat, SAT, Lecture Notes in Computer Science, Springer 5584 (2009) 467–480.

[47] F. Heras, J. Larrosa, A. Oliveras, Minimaxsat: a new weighted max-sat solver, In International Conference on Theory and Applications of Satisfiability Testing (2007) 41–55.

[48] H. Lin, K. Su, C. M. Li, Within-problem learning for efficient lower bound computation in max-sat solving, In International Conference on Theory and Applications of Satisfiability Testing (2008) 351–356.

[49] D. L. Berre, Sat4jmaxsat, a satisfability library for java: http://sat4j.org/(2008).

[50] V. M. Manquinho, J. P. M. Silva, J. Planes, Algorithms for weighted boolean optimization, In Procedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (2009) 495–508.

[51] C. Ansótegui, M. L. Bonet, J. Levy, Solving (weighted) partial maxsat through satisfiability testing, In Procedings of the 20th International Con-

ference on Theory and Applications of Satisfiability Testing 5584 (2009) 427–440.

[52] C. Anstegui, M. L. Bonet, J. Levy, A new algorithm for weighted partial maxsat, In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence.

[53] E. Amaldi, From finding maximum feasible subsystems of linear systems to feedforward neural network design (1994).

[54] E. Amaldi, M. Bruglieri, G. Casale, A two-phase relaxation-based heuristic for the maximum feasible subsystem problem, Computers and Operations Research (2008) 1465–1482.

[55] O. Mangasarian, Misclassification minimization, Journal of Global Optimization (1994) 309–323.

[56] M. Pfetsch, Branch and cut for the maximum feasible subsystemproblem, SIAM Journal on Optimization (2008) 21–38.

[57] J. W. Chinneck, E. Dravnieks, Locating minimal infeasible constraint sets in linear programs, ORSA Journal on Computing (1991) 157–168.

[58] R. Bakker, F. Dikker, F. Tempelman, P. Wogmim, Diagnosing and solving over-determined constraint satisfaction problems, International Joint Conference on Artificial Intelligence (1993) 276–281.

[59] M. Tamiz, S. J. Mardle, D. F. Jones, Resolving inconsistency in infeasible linear programmes, Tech. rep. (1995).

[60] M. Tamiz, S. J. Mardle, D. F. Jones, Detecting iis in infeasible linear programmes using techniques from goal programming, Computers and Operations Research (1996) 113–119.

[61] C. L. G. W. Noreen Jamil, Johannes Müller, Extending linear relaxation for user interface layout, Proceedings of 24th International Conference on Tools with Artificial Intelligence (ICTAI) (2012) 1–8.

[62] O. Guieu, J. W. Chinneck, Analyzing infeasible mixed-integer and integer linear programs, INFORMS Journal on Computing (1999) 63–77.